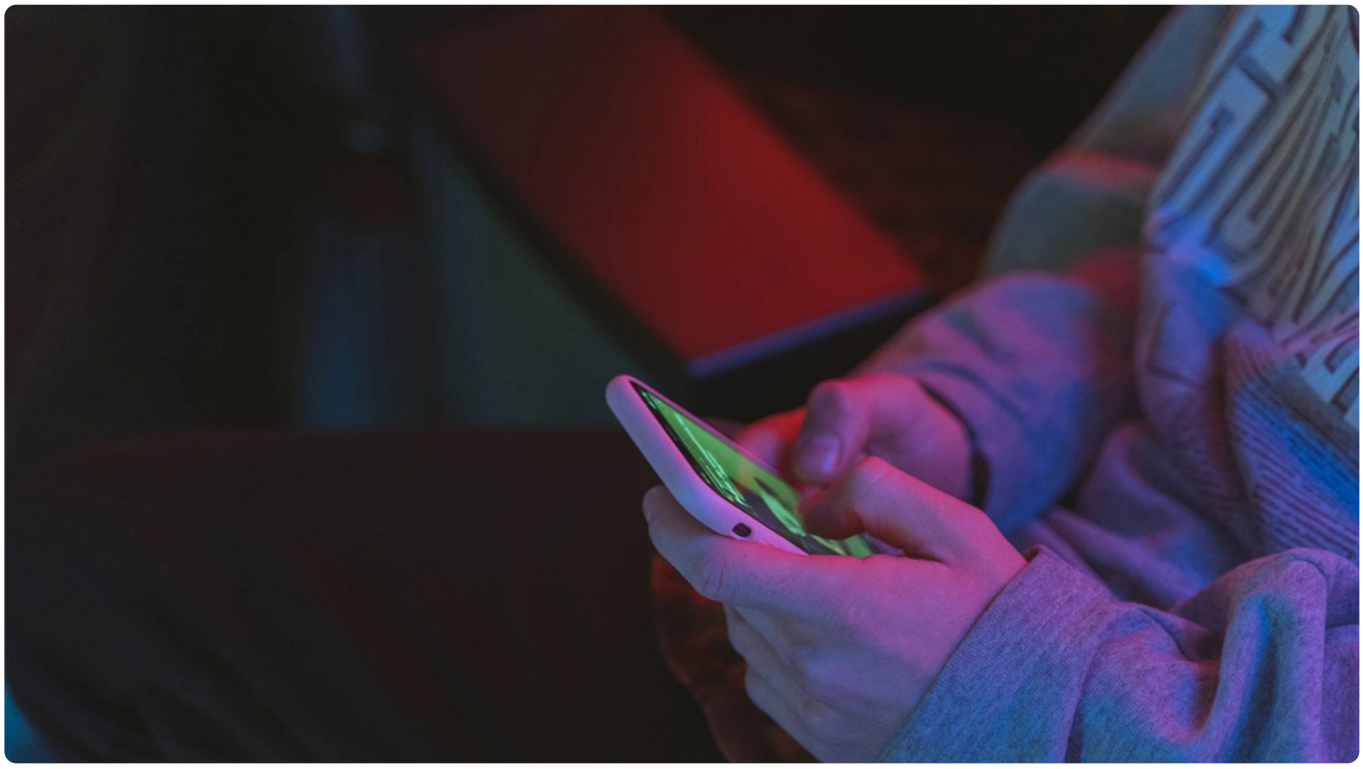
09 | 交友系统设计：哪种地理空间邻近算法更快？

2022-03-07 李智慧

《李智慧 · 高并发架构实战课》 课程介绍 >



讲述：李智慧
时长 16:35 大小 15.19M

你好，我是李智慧。

交友与婚恋是人们最基本的需求之一。随着互联网时代的不断发展，移动社交软件已经成为了人们生活中必不可少的一部分。然而，熟人社交并不能完全满足年轻人的社交与情感需求，于是陌生人交友平台悄然兴起。

我们决定开发一款基于地理位置服务（LBS）的应用，为用户匹配邻近的、互相感兴趣的好友，应用名称为“Liao”。

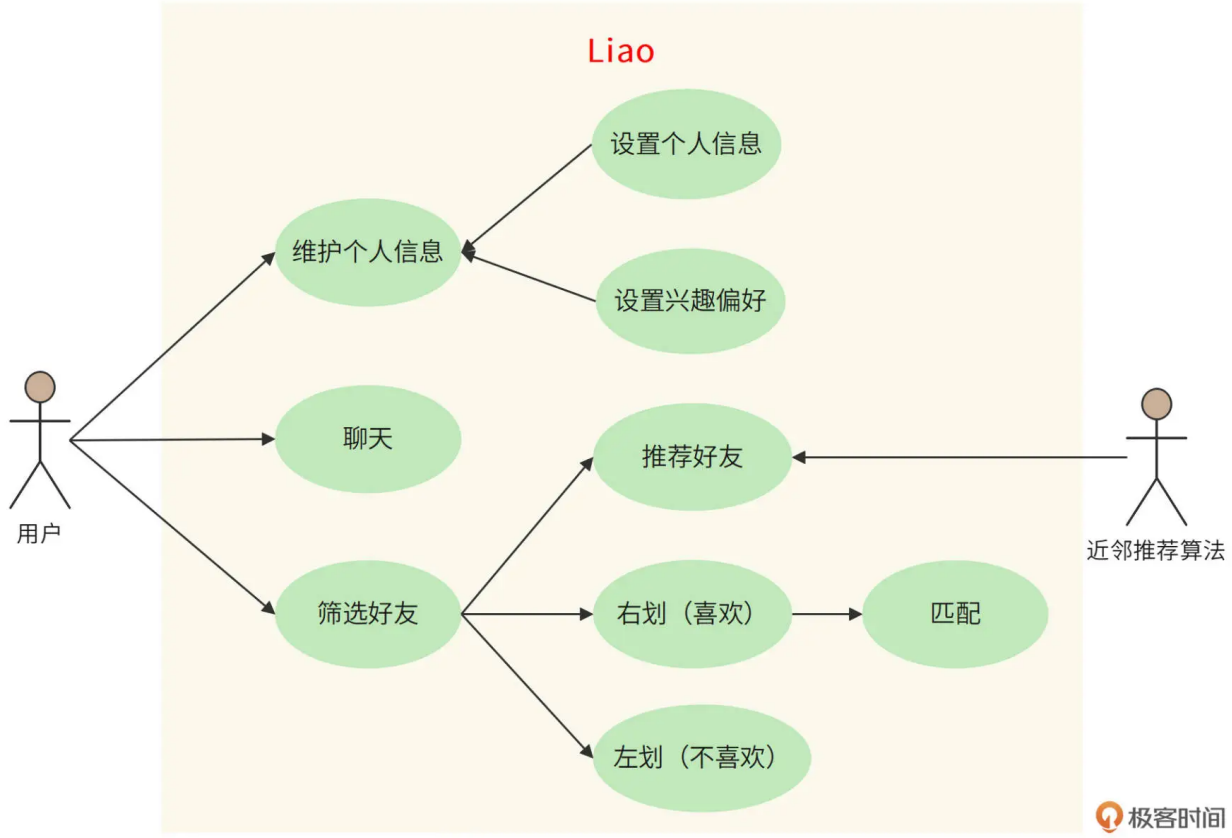
Liao 面临的技术挑战包括：面对海量的用户，如何为其快速找到邻近的人，可以选择的地理空间邻近算法有哪些？Liao 如何在这些算法中选择出最合适的那个？

需求分析

Liao 的客户端是一个移动 App，用户打开 App 后，上传、编辑自己的基本信息，然后系统（推荐算法）根据其地理位置和个人信息，为其推荐位置邻近的用户。用户在手机上查看对方的照片和资料，如果感兴趣，希望进一步联系，就向右滑动照片；如果不感兴趣，就向左滑动照片。

如果两个人都向右滑动了对方，就表示他们互相感兴趣。系统就通知他们配对成功，并为他们开启聊天功能，可以更进一步了解对方，决定是否建立更深入的关系。

Liao 的功能用例图如下。

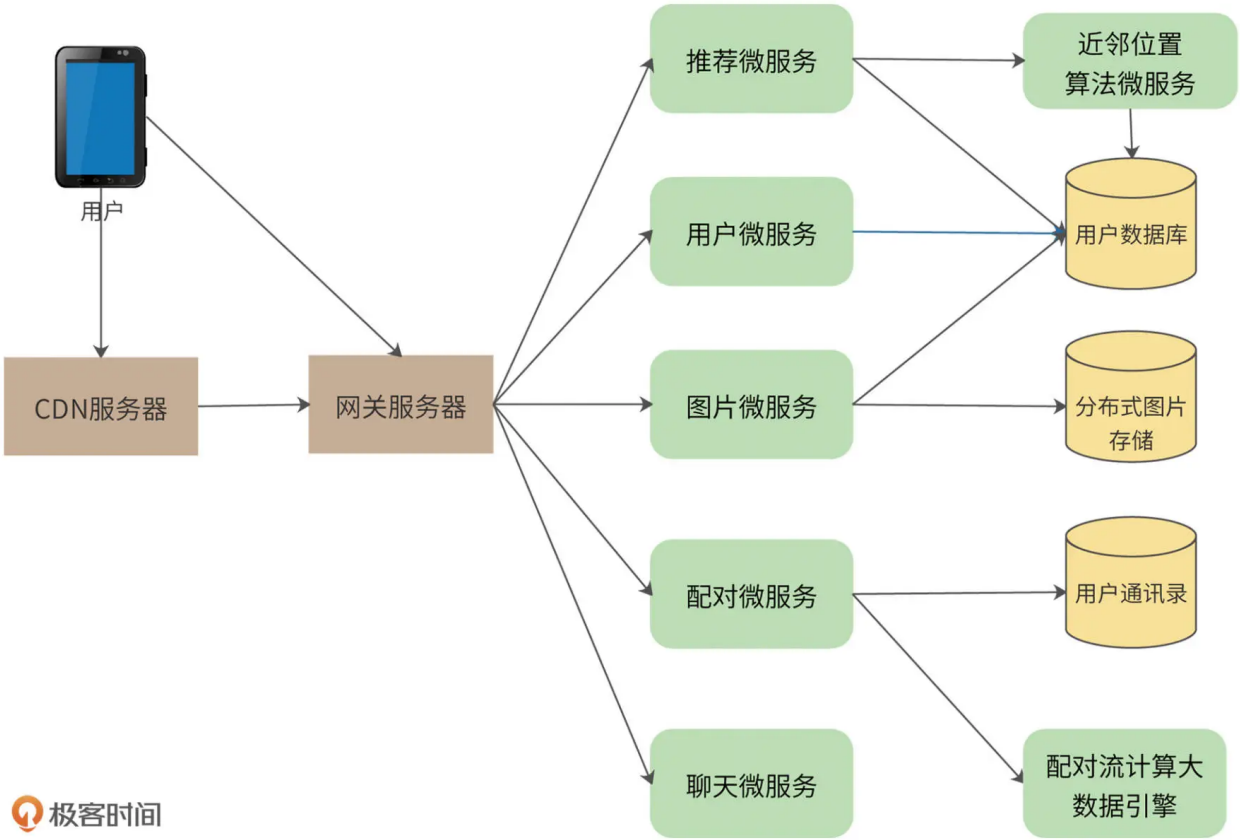


用户规模分析

Liao 的目标用户是全球范围内的中青年单身男女，预估目标用户超过 10 亿，系统按 10 亿用户进行设计。

概要设计

Liao 的系统架构采用典型的**微服务架构**设计方案，用户通过网关服务器访问具体的微服务，如下图。



由上图可知，首先，用户所有请求都通过统一的**网关服务器**处理。网关服务器负责限流、防攻击、用户身份识别及权限验证、微服务调用及数据聚合封装等，而真正的业务逻辑则通过访问微服务来完成。Liao 的关键微服务有：用户微服务、图片微服务、配对微服务、聊天微服务、推荐微服务、邻近算法微服务等。Liao 的网关预计将承担每天百亿次规模的访问压力。

用户微服务管理用户的个人信息、兴趣爱好以及交友偏好等，此外也负责用户登录服务，只有登录用户才能访问系统。因为需要存储十亿条用户数据，所以用户数据库采用分片的 MySQL 数据库。

图片微服务用于管理用户照片，提供用户照片存储及展示的功能。Liao 需要存储的图片数大约几百亿张。我们使用 Nginx 作为图片服务器，图片服务器可以线性扩容，每写满一台服务器（及其 Slave 服务器），就继续写入下一台服务器。服务器 IP、图片路径则记录在用户数据库中。同时，购买 CDN 服务，缓存热门的用户照片。

配对微服务负责将互相喜欢的用户配对，通知用户，并加入彼此的通讯录中。用户每次右划操作都调用该微服务。系统设置一个用户每天可以喜欢（右划）的人是有上限的，但是，对于活跃用户而言，长期积累下来，喜欢的人的数量还是非常大的，因此配对微服务会将数据发送给一个流式大数据引擎进行计算。

推荐微服务负责向用户展示其可能感兴趣的、邻近的用户。因此，一方面，推荐微服务需要根据用户操作、个人兴趣、交友偏好调用协同过滤等推荐算法进行推荐，另一方面必须保证推荐的用户在当前用户的附近。

详细设计

详细设计主要关注邻近位置算法，也就是，如何根据用户的地理位置寻找距其一定范围内的其他用户。

我们可以通过 Liao App 获取用户当前经、纬度坐标，然后根据经、纬度，计算两个用户之间的距离，距离计算公式采用半正矢公式：

$$
\begin{aligned}
d &= 2r\arcsin\left(\sqrt{\operatorname{hav}(\varphi_2-\varphi_1)+\cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2-\lambda_1)}\right) \\
&= 2r\arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2-\varphi_1}{2}\right)+\cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2-\lambda_1}{2}\right)}\right)
\end{aligned}
$$

其中 r 代表地球半径，φ 表示纬度，λ 表示经度。

但是，当我们有 10 亿用户的时候，如果每次进行当前用户匹配都要和其他所有用户做一次距离计算，然后再进行排序，那么需要的计算量至少也是千亿级别，这样的计算量是我们不能承受的。通常的空间邻近算法有以下 4 种，我们一一进行分析，最终选择出最合适的方案。

SQL 邻近算法

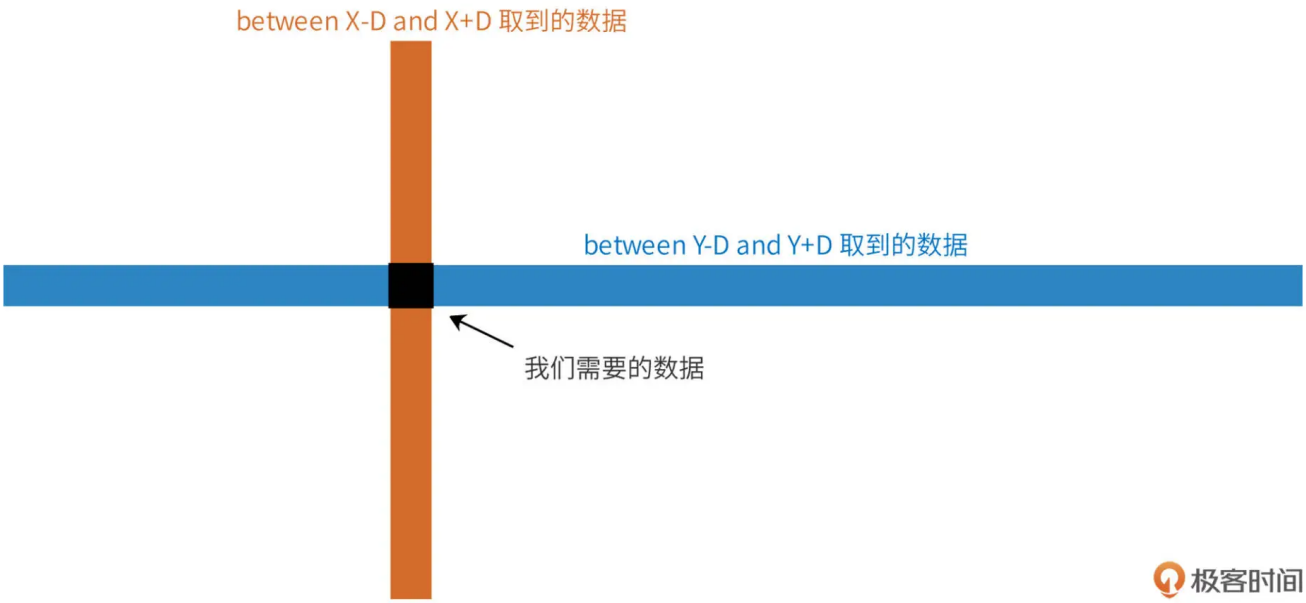
我们可以将用户经、纬度直接记录到数据库中，纬度记录在 latitude 字段，经度记录在 longitude 字段，用户当前的纬度和经度为 X，Y，如果我们想要查找和当前用户经、纬度距离 D 之内的其他用户，可以通过如下 SQL 实现。

复制代码

```
1 select * from users where latitude between X-D and X+D and longtitude between
```

这样的 SQL 实现起来比较简单，但是如果有十亿用户，数据分片在几百台服务器上，SQL 执行效率就会很低。而且我们用经、纬度距离进行近似计算，在高纬度地区，这种近似计算的偏差还是非常大的。

同时“between X-D and X+D”以及“between Y-D and Y+D”也会产生大量中间计算数据，这两个 betwen 会先返回经度和纬度各自区间内的所有用户，再进行交集 and 处理，如下图。



我们的用户量非常大，而计算邻近好友又是一个非常高频的访问，同时，分片数据库进行集合计算需要在中间代理服务器或应用程序服务器完成计算，因此，这样的交集计算带来计算负载压力是我们的系统完全不能承受的。所以这个方案可以被放弃。

地理网格邻近算法

为了减少上述交集计算使用的中间数据量，我们将整个地球用网格进行划分，如下图。

极客时间

事实上，我们划分的网格远比图中示意的要密集得多，赤道附近，经、纬度方向每 10 公里一个网格。

这样每个用户必然会落入到一个网格中，我们在用户表中记录用户所在的网格 ID（gridID），然后借助这个字段进行辅助查找，将查找范围限制在用户所在的网格（gridIDx0）及其周围 8 个网格（gridIDx1 ~ gridIDx8）中，可以极大降低中间数据量，SQL 如下。

```
1 select * from users where latitude between X-D and X+D and longtitude between
```

复制代码

这条 SQL 要比上面 SQL 的计算负载压力小得多，但是对于高频访问的分片数据库而言，用这样的 SQL 进行邻近好友查询依然是不能承受的，同样距离精度也不满足要求。

但是基于这种网格设计思想，我们发现，我们可以不通过数据库就能实现邻近好友查询：我们可以**将所有的网格及其包含的用户都记录在内存中**。当我们进行邻近查询时，只需要在内存中计算用户及其邻近的 8 个网格内的所有用户的距离即可。

我们可以估算下所有用户经、纬度都加载到内存中需要的内存量：$1G \times 3 \times 4B = 12GB$（用户 ID、经度、纬度，都采用 4 个字节编码，总用户数 1G）。这个内存量是完全可以接受的。

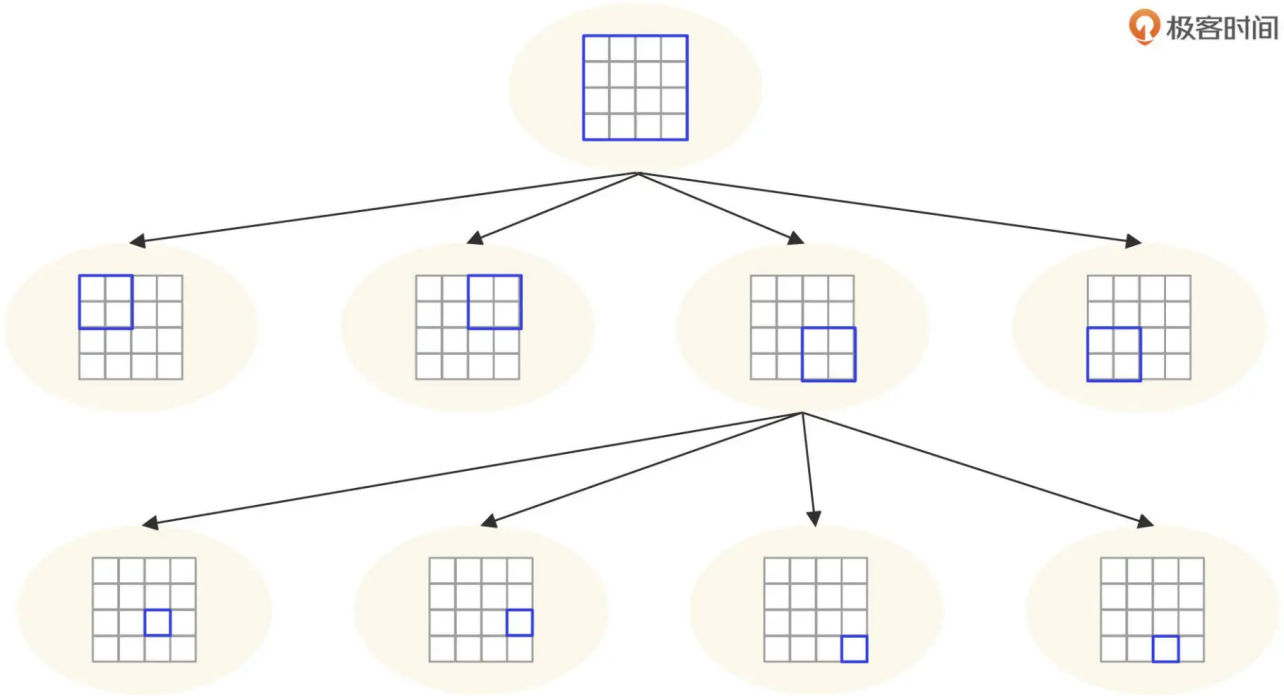
实际上，通过恰当地选择网格的大小，我们不停访问当前用户位置周边的网格就可以由近及远不断得到邻近的其他用户，而不需要再通过 SQL 来得到。那么如何选择网格大小？如

何根据用户位置得到其所在的网格？又如何得到当前用户位置周边的其他网格呢？我们看下实践中更常用的动态网格和 GeoHash 算法。

动态网格算法

事实上，不管如何选择网格大小，可能都不合适。因为在陆家嘴即使很小的网格可能就包含近百万的用户，而在可可西里，非常大的网格也包含不了几个用户。

因此，我们希望能够动态设定网格的大小，如果一个网格内用户太多，就把它分裂成几个小网格，小网格内如果用户还是太多，继续分裂更小的网格，如下图。



这是一个四叉树网格结构，开始的时候整个地球只有一个网格，当用户增加，超过阈值（500 个用户）的时候，就分裂出 4 个子树，4 个子树对应父节点网格的 4 个地理子网格。同时，将用户根据位置信息重新分配到 4 个子树中。同样，如图中所示，如果某个子树中的用户增加，超过了阈值，该子树继续分裂成 4 个子树。

因此，我们可以将全球用户分配在这样一个 4 叉树网格结构中，所有的用户都必然在这个 4 叉树的叶子节点中，而且每个节点内包含的用户数不超过 500 个。那么，陆家嘴的网格可能就会很小，而可可西里的网格就会很大，太平洋对应的网格可能有几千公里。

当给定当前用户的经、纬度，查询邻近用户的时候，首先从根节点开始查找，如果根节点就是叶子节点，那么直接遍历根节点中的所有用户，计算距离即可。如果根节点不是叶子节点，那么根据给定的经、纬度判断其在网格中的位置，左上、右上、右下、左下，4 个位置，顺序对应 4 个子树，根据网格位置访问对应的子树。如果子树是叶子节点，那么在叶子节点中查找，如果不是叶子节点，继续上面的过程，直到叶子节点。

上面的过程只能找到当前用户所在网格的好友，如何查找邻近网格的其他用户呢？事实上，我们只需要将 4 叉树所有的叶子节点顺序组成一个双向链表，每个节点在链表上的若干个前驱和后继节点正好就是其地理位置邻近的节点。

动态网格也叫 4 叉树网格，在空间邻近算法中较为常用，也能满足 Liao 的需求。但是编程实现稍稍有点麻烦，而且如果网格大小设计不合适，导致树的高度太高，每次查找需要遍历的路径太长，性能结果也比较差。我们再看下性能和灵活性更好的 GeoHash 算法。

GeoHash 算法

除了动态网格算法，GeoHash 事实上是另外一种变形了的网格算法，同时也是 Redis 中 Geo 函数使用的算法。GeoHash 是将网格进行编码，然后根据编码进行 Hash 存储的一种算法。

经、纬度数字的不同精度，意味着经、纬度的误差范围，比如保留经、纬度到小数点后第 1 位，那么误差范围最大可能会达到 11 公里（在赤道附近）。也就是说，小数点后 1 位精度的经、纬度，其覆盖范围是一个 11km * 11km 的网格。

那么，我们用小数点后 1 位精度的经、纬度做 key，网格内的用户集合做 value，就可以构建一个 Hash 表的 <key, value> 对。通过查找这个 KV 对及其周围 8 个网格的 KV 对，计算这些 value 内所有用户和当前用户的距离，就可以找到邻近 11 公里内的所有用户。

实践中，redis 的 GeoHash 并不会直接用经、纬度做 key，而是采用一种基于 Z 阶曲线的编码方式，将二维的经、纬度，转化为一维的二进制数字，再进行 base32 编码，具体过程如下。

首先，分别针对经度和纬度，求取当前区间（对于纬度而言，开始的区间就是[-90, 90], 对于经度而言，开始区间就是[-180, 180]）的平均值，将当前区间分为两个区间。然后用用户的经、纬度和区间平均值进行比较，用户经、纬度必然落在两个区间中的一个，如果大

于平均值，那么取 1，如果小于平均值，那么取 0。继续求取当前区间的平均值，进一步将当前区间分为两个区间。如此不断重复，可以在经度和纬度方向上，得到两个二进制数。这个二进制数越长，其所在的区间越小，精度越高。

下图表示经、纬度 <43.60411, -5.59041> 的二进制编码过程，最终得到纬度 12 位编码，经度 13 位编码。

Latitude code 101111001001						
bit position	bit value	min	mid	max	mean value	maximum error
0	1	-90.000	0.000	90.000	45.000	45.000
1	0	0.000	45.000	90.000	22.500	22.500
2	1	0.000	22.500	45.000	33.750	11.250
3	1	22.500	33.750	45.000	39.375	5.625
4	1	33.750	39.375	45.000	42.188	2.813
5	1	39.375	42.188	45.000	43.594	1.406
6	0	42.188	43.594	45.000	42.891	0.703
7	0	42.188	42.891	43.594	42.539	0.352
8	1	42.188	42.539	42.891	42.715	0.176
9	0	42.539	42.715	42.891	42.627	0.088
10	0	42.539	42.627	42.715	42.583	0.044
11	1	42.539	42.583	42.627	42.605	0.022

Longitude code 0111110000000						
bit position	bit value	min	mid	max	mean value	maximum error
0	0	-180.000	0.000	180.000	-90.000	90.000
1	1	-180.000	-90.000	0.000	-45.000	45.000
2	1	-90.000	-45.000	0.000	-22.500	22.500
3	1	-45.000	-22.500	0.000	-11.250	11.250
4	1	-22.500	-11.250	0.000	-5.625	5.625
5	1	-11.250	-5.625	0.000	-2.813	2.813
6	0	-5.625	-2.813	0.000	-4.219	1.406
7	0	-5.625	-4.219	-2.813	-4.922	0.703
8	0	-5.625	-4.922	-4.219	-5.273	0.352
9	0	-5.625	-5.273	-4.922	-5.449	0.176
10	0	-5.625	-5.449	-5.273	-5.537	0.088
11	0	-5.625	-5.537	-5.449	-5.581	0.044
12	0	-5.625	-5.581	-5.537	-5.603	0.022

得到两个二进制数后，再将它们合并成一个二进制数。合并规则是，从第一位开始，奇数位为经度，偶数位为纬度，上面例子合并后的结果为 01101 11111 11000 00100 00010，共 25 位二进制数。

将 25 位二进制数划分成 5 组，每组 5 个二进制数，对应的 10 进制数是 0-31，采用 Base32 编码，可以得到一个 5 位字符串，Base32 编码表如下。

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base 32	0	1	2	3	4	5	6	7	8	9	b	c	d	e	f	g
Decimal	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Base 32	h	j	k	m	n	p	q	r	s	t	u	v	w	x	y	z

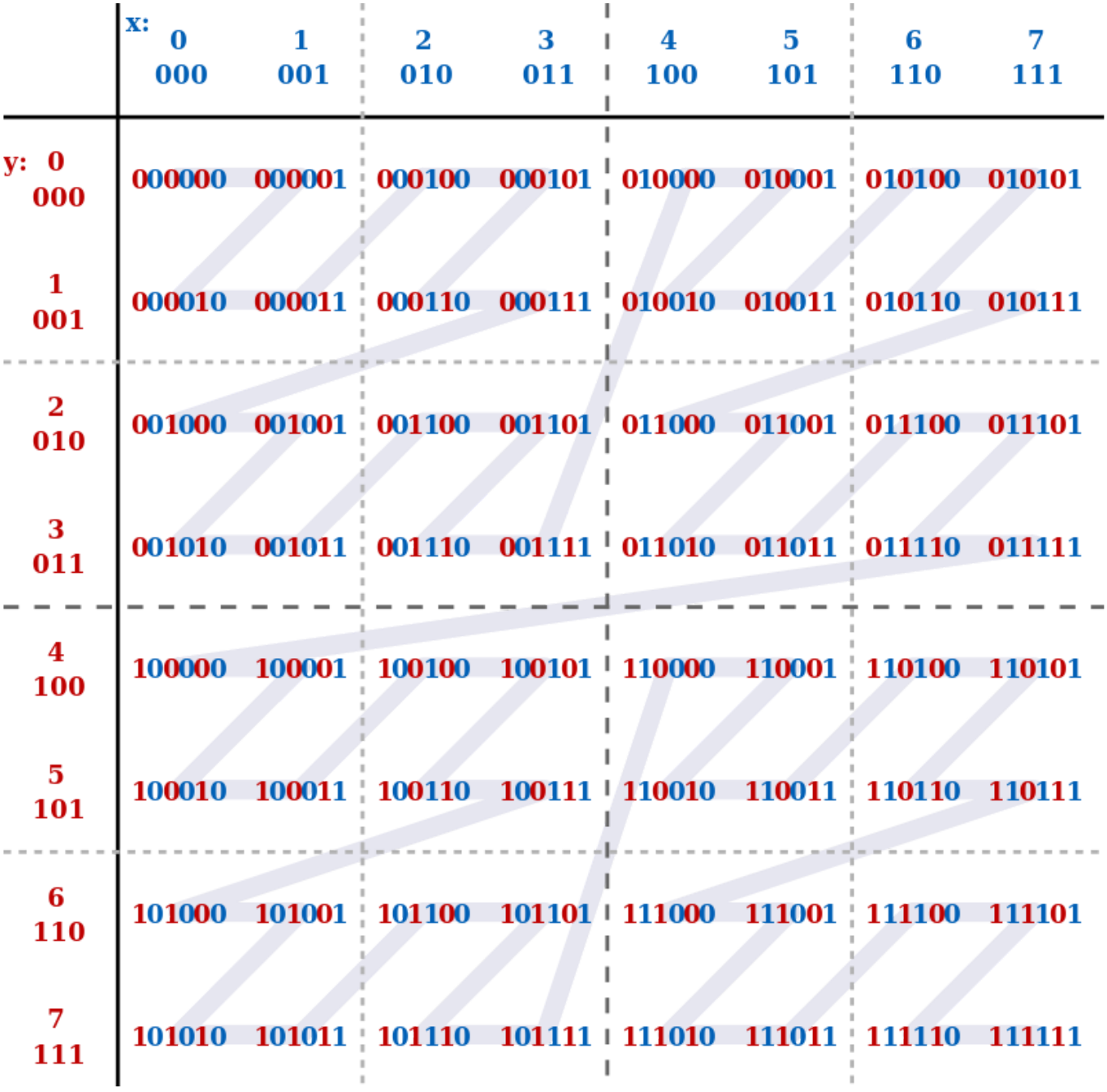
编码计算过程如下。

$$
\begin{aligned}
[e]_{32ghs} &= [13]_{10} = [01101]_2 \\
[z]_{32ghs} &= [31]_{10} = [11111]_2 \\
[s]_{32ghs} &= [24]_{10} = [11000]_2 \\
[4]_{32ghs} &= [4]_{10} = [00100]_2 \\
[2]_{32ghs} &= [2]_{10} = [00010]_2.
\end{aligned}
$$

最后得到一个字符串“ezs42”，作为 Hash 表的 key。25 位二进制的 GeoHash 编码，其误差范围大概 2.4 公里，即对应一个 $2.4km \times 2.4km$ 的网格。网格内的用户都作为 value 放入到 Hash 表中。

一般说来，通过选择 GeoHash 的编码长度，实现不同大小的网格，就可以满足我们邻近交友的应用场景了。但是在 Redis 中，需要面对更通用的地理位置计算场景，所以 Redis 中的 GeoHash 并没有用 Hash 表存储，而是用跳表存储。

Redis 使用 52 位二进制的 GeoHash 编码，误差范围 0.6 米。Redis 将编码后的二进制数按照 Z 阶曲线的布局，进行一维化展开。即将二维的经、纬度上的点，用一条 Z 型曲线连接起来，Z 阶曲线布局示例如下图。



事实上，所谓的 Z 阶曲线布局，本质其实就是基于 GeoHash 的二进制排序。将这些经过编码的 2 进制数据用跳表存储。查找用户的时候，可以快速找到该用户，沿着跳表前后检索，得到的就是邻近的用户。

Liao 的最终算法选择

Liao 的邻近算法最终选择使用 Hash 表存储的 GeoHash 算法，经度采用 13bit 编码，纬度采用 12bit 编码，即最后的 GeoHash 编码 5 个字符，每个网格 $2.4km \times 2.4km \approx 5km^2$，将整个地球分为 $2^{25} \approx 3300$万 个网格，去掉海洋和几乎无人生存的荒漠极地，需要存储的 Hash 键不到 500 万个，采用 Hash 表存储。Hash 表的 key 是 GeoHash 编码，value 是一个 List，其中包含了所有相同 GeoHash 编码的用户 ID。

查找邻近好友的时候，Liao 将先计算用户当前位置的 GeoHash 值（5 个字符），然后从 Hash 表中读取该 Hash 值对应的所有用户，即在同一个网格内的用户，进行匹配，将满足匹配条件的对象返回给用户。如果一个网格内匹配的对象数量不足，计算周围 8 个网格的 GeoHash 值，读取这些 Hash 值对应的用户列表，继续匹配。

小结

算法是软件编程中最有技术挑战性，也最能考验一个人编程能力的领域。所以很多企业面试的时候特别喜欢问算法类的问题，即使这些算法和将来的工作内容关系不大，面试官也可以凭借这些问题对候选人的专业能力和智力水平进行评判，而且越是大厂的面试越是如此。

架构和算法通常是一个复杂系统的一体两面，架构是关于整体系统是如何组织起来的，而算法则是关于核心功能如何处理的。我们专栏大多数案例也都体现了这种一体两面，很多案例设计都有一两个核心算法，比如短 URL 生成与预加载算法、缩略图生成与推荐算法、本篇的空间邻近算法以及下一篇要讲的倒排索引与 PageRank 算法，都展现了这一点。

一个合格的架构师除了要掌握系统的整体架构，也要能把握住这些关键的算法，才能在系统的设计和开发中做到心中有数、控制自如。

思考题

本文的设计聚焦在邻近算法上，所以忽略了常规的 TPS、带宽负载、存储空间等性能指标，你能否估计下这些性能指标？

欢迎在评论区分享你的思考，或者提出对这个设计文档的评审意见，我们共同进步。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | 秒杀系统设计：你的系统可以应对万人抢购盛况吗？

下一篇 10 | 搜索引擎设计：信息搜索怎么避免大海捞针？

精选留言 (10)

写留言

peter

2022-03-07

请教老师几个问题啊： Q1：网关可以是最外层吗？ 本文中的架构图，请求直接到网关，网关在最外层，这可以吗？ 微服务开发，现在一般用springcloud,其网关是spring gateway,可以让请求直接到spring gateway吗？（有的说最好不要让spring gateway在最外面，其前面最好加一个nginx）。 Q2：每个微服务用自己独立的DB吗？ 本文中的架构图，有几个微服务共用了一个DB。但有的说"微服务之间相互独立，DB也相互独立，即一个微服务...

展开

作者回复: 1 网关前应该有负载均衡，文中简化省略了。spring+nginx是正确用法

2 微服务可以共用db，db独立的主要考虑是为了降低负载压力。

3 java和redis不会在一个机器上，微服务架构下尤其不会。PS：文中算法用redis，但是并没有用redis的geo函数。

4 是的

5 -5.5是经度longitude，是0。文中经、纬度的叫法是人们习惯叫法；二元组<43.60411, -5.59041 > 写法，纬度在前，经度在后，是经纬度的习惯写法。

1

周文童
2022-03-11

沈剑老师也分享过几乎一致的解决方案，果然高手所见略同啊

zhaobk
2022-03-11

老师好，请问，postgresql 有个gis插件，是否可以考虑用于计算查找临近的好友？

作者回复: 不了解这个插件，感谢提示。不过数据库上的操作负载压力必然是满足不了我们这个场景需求的，太笨重了。

共 3 条评论 >

赵常权
2022-03-10

geohash 维度第7 9 10 行 43.60411大于纬度平均值 为什么是0呢

逐风
2022-03-09

geoHash有办法对临近的用户进行距离排序吗

作者回复: Z阶曲线本身就是一个排序

qc
2022-03-09

10亿用户这个规模下，我们应该采用什么存储方案记录用户之间的好友关系呢？

作者回复: 暴力一点的话，在用户表里，好友用一个字段就可以记录了，关注好友有上限，一般2000个，好友id用整型，字段最大长度8k。

2022-03-08

GeoHash的算法确实很巧妙，但如果按这样存储，如何按不同的距离过滤用户呢？要生成不同网格大小的GeoHash码吗？感觉应该有更好的方法。

作者回复: 用GeoHash码的长度控制网格大小，网格大小是固定的。

aoe

2022-03-08

原来距离是通过网格计算出来的！

2022-03-07

目标用户10亿 预估日活10%=1亿，月活20%=2亿 TPS：每个用户，没天滑动10个目标用户。1亿*10=10亿。 互相发送消息20条。1亿*2*20=40亿。 TPS=50亿/86400 假设同时在线用户，预估5%=500万 带宽：由于文字占用的带宽极小，个人认为，可忽略不计。主要占用带宽的资源为图片资源。 平均每个用户，发送5张图片表情包。其中2张新增，3张服务器保存。一张表情包100kb。 假设同时护发图片的用户10%=50万。带宽=50万*2*...

展开

作者回复: 赞，思路很清晰

1

Geek_7347cf

2022-03-07

事实上用户是会自主行走的 ，当用户地理空间发生移动时，如何更新数据呢

作者回复: 用户打开App的时候，更新数据：删除旧的位置hash key的用户，添加到新的位置hash key。所以应用还需要记录键值对：用户->位置。

1